The Linley Group

MICROPROCESSOR report

Insightful Analysis of Processor Technology

NVIDIA'S FIRST CPU IS A WINNER

Denver Uses Dynamic Translation to Outperform Mobile Rivals

By Linley Gwennap (August 18, 2014)

Nvidia's first CPU design, code-named Denver, delivers ARMv8 compatibility and high performance using an unusual technique: dynamic instruction translation. As explained by CPU architect Darrell Boggs at this week's Hot Chips conference, Denver can execute ARMv8 code in two ways: either natively at a peak rate of two instructions per cycle or by using what Boggs calls dynamic code optimization to achieve a peak rate of seven micro-ops per cycle. Nvidia's initial benchmark testing shows the 64-bit CPU outperforming all other announced ARM cores and even matching the performance of low-end versions of Intel's Haswell CPU.

The first product to use Denver is Tegra K1-64, which is designed for tablets, high-end smartphones, automotive in-dash systems, and similar applications. This chip is similar to Tegra K1-32 but replaces that chip's four 2.3GHz Cortex-A15s with two Denver CPUs running at up to 2.5GHz. The two chips are pin compatible and have roughly the same thermal envelope, enabling Tegra customers to drop in the K1-64 as soon as it's ready. The chip is already sampling and is scheduled to enter production in 4Q14.

To deliver high performance in a mobile power profile, Nvidia moved much of the instruction-scheduling function into software using dynamic translation. This approach is similar to what Transmeta did more than a decade ago (see *MPR 2/14/00,* "Transmeta Breaks x86 Low-Power Barrier"). Unlike that design, Denver includes some hardware-based instruction decoders; thus, it need only translate frequently used routines, reducing the translation overhead.

Nvidia has based previous Tegra processors, including the K1-32, on licensed ARM cores, making Denver its first attempt at internal CPU design. (Of course, the company has designed its own graphics processors for years.) It's not bad for a first try: Tegra K1-64 outperforms all competing high-end mobile processors, particularly on single-threaded benchmarks. The K1-64 is the first high-end ARMv8 mobile processor that is generally available to OEMs (unlike Apple's in-house processor), giving Nvidia another edge over its 32-bit rivals.

Dynamic Instruction Translation

When a program begins executing on Denver, the instructions pass through the decoders and into the execution units, as in any other CPU. In addition to its normal duties, the branch unit counts the number of times each routine (represented by a branch target address) is encountered. When this count passes a certain threshold, the CPU triggers a micro-interrupt that jumps to a firmware routine called the optimizer, as Figure 1 shows.

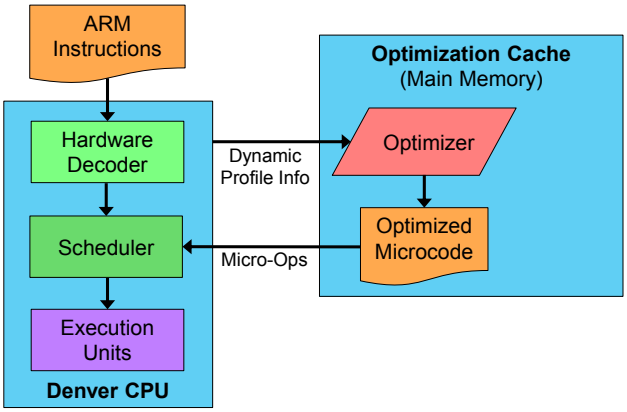

Figure 1. Code translation in Nvidia's Denver CPU. The optimizer converts ARMv8 instructions into native microcode, storing the result in a main-memory block called the optimization cache.

The optimizer analyzes the indicated ARM code and translates it into the CPU's native microcode (micro-ops). Each ARM instruction converts to 1.8 micro-ops. ARM's own CPUs also use microcode but generate about 1.1 or 1.2 micro-ops per instruction; Denver's micro-ops are simpler and more general purpose than in ARM's designs. The Denver micro-ops have a variable length but average about 5.7 bytes. Taking into account Thumb encodings, the average ARM instruction contains 3.1 bytes and converts to 10.3 bytes of micro-ops, increasing code size by 3.3x. (These figures are based on SPECint2000 and will vary somewhat by application.)

As in a VLIW design, micro-ops are grouped into bundles such that all micro-ops in a bundle can be executed in the same cycle. To minimize code expansion, the optimizer suppresses any unused slots, resulting in variable-length bundles. Each bundle can contain up to seven micro-ops but is capped at 32 bytes. Because of this cap, even a maximum-size bundle typically contains five or six micro-ops. The average bundle size is three or four micro-ops (18 to 24 bytes).

Befitting its name, the optimizer does far more than simple translation. To maximize the use of slots in each bundle, it reorders micro-ops and renames registers to work around dependencies. Rather than simply optimizing a basic block, it unrolls loops and works past branches using information from the branch predictor to determine the most likely direction of each branch. It can even convert subroutines to inline code.

An optimized routine can encompass up to 1,000 ARM instructions—a far larger window than any out-of-order (OOO) scheduler implemented in hardware. The optimizer may stop sooner, however, if it runs out of resources, such as rename registers or store-buffer entries. It must also stop if it encounters an instruction that changes the privilege level or performs other system functions (e.g., cache flush).

The translated microcode is stored in a 128MB reserved area of main memory that Nvidia calls the optimization cache. The instruction-fetch unit includes a translation table that converts the address of an ARM routine to the address of the corresponding microcode routine; each time it encounters the translated routine, the CPU automatically fetches from the optimization cache instead of the original ARM code. As more routines are translated, the optimizer can link them together, allowing the CPU to branch directly from one microcode routine to another.

The optimization cache is protected such that only microcode (optimized) routines can write to it, providing security against malicious or poorly written code that attempts to overwrite the optimizer or optimized code sequences. For this reason, and to improve performance, the optimizer is preconverted to microcode and stored in the optimization cache. Unlike processors that perform instruction scheduling in hardware, this approach allows Nvidia to easily update the optimizer to improve performance or fix bugs.

The Optimizer in Action

Figure 2 shows a profile of a typical program running on Denver. As the program starts, the CPU executes mainly ARM instructions (the yellow bars), but it quickly identifies "hot" routines and begins translating them into microcode that is later directly executed (the green bars). After just 2% of the total execution time, the CPU is largely executing microcode. Partway through the run, this particular program deploys a set of new routines, causing a brief surge in ARM instructions; again, however, these instructions are quickly translated into microcode. The CPU then executes nearly 100% microcode until it encounters the teardown code at the end of the program.

The red bars show the overhead of the optimizer itself. Even during the few periods with extensive translation, the optimizer consumes fewer than 20% of the cycles; most of the time, it uses less than 10%, and it completely disappears for more than half the test. Over the course of this particular program, the optimization overhead is only 3.2%. For highly repetitive programs, including some SPECfp2000 components, the optimization overhead for the entire program is nearly zero. On the other hand, complex programs with many functions will have considerably more overhead.

Even after a routine is translated, the CPU can again invoke the optimizer if the behavior of a branch changes over time. This situation can happen if conditions change during a program's execution (for example, the scene in a game program changes from day to night). Specifically, when the branch predictor changes the prediction for a branch, it will invoke the optimizer; the optimizer determines which optimized routine contains the branch and generates new microcode for that routine. This type of reoptimization may be responsible for the small burst of optimizer activity about halfway through the benchmark run in Figure 2.

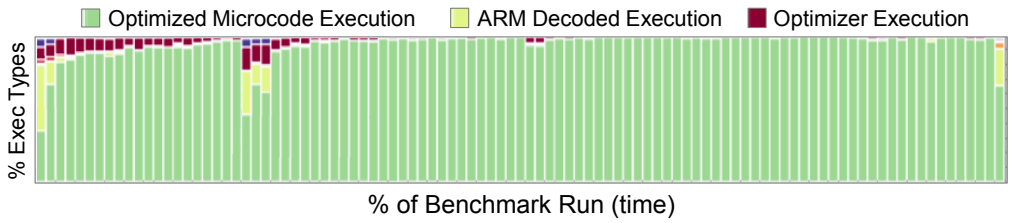

Figure 2. Program execution on the Denver CPU. When running the "crafty" component of the SPECint2000 benchmark, the CPU initially executes mostly ARM instructions but soon switches almost entirely to microcode. (Source: Nvidia)

Front End Handles Two Instruction Types

The Denver microarchitecture combines a fairly simple front end with a wide set of execution units, as Figure 3 shows. The instruction cache is 128KB, four times the usual size for Cortex-A15 and other popular ARM CPUs; the 3x code expansion of the micro-ops forced Nvidia to adopt this larger cache. The instruction TLB has 128 entries and is four-way associative. Each cycle, the I-cache feeds 32 bytes into the fetch queue. This bandwidth matches the maximum bundle length.

In ARM mode, the fetch queue feeds eight bytes per cycle into the ARM decoder, which decodes two ARM instructions into micro-ops. The decoder handles both ARMv8 instructions and legacy ARMv7 (Aarch32) apps. If possible, the decoder combines the micro-ops into a single bundle; if the two instructions have dependencies or resource conflicts, it creates two separate bundles.

In native mode, the fetch unit extracts a single variable-length bundle from the fetch queue and feeds it into a microcode decoder. Although the micro-ops are stored in memory in an expanded form, some fields are still multiplexed to reduce code bloat. The microcode decoder demultiplexes these fields into a series of one-hot signals that indicate, for example, which function unit will execute each micro-op. This advance work speeds the scheduler's operation.

As noted above, a maximum-size bundle most often contains five or six micro-ops, which represent on average three ARM instructions. Thus, even assuming the optimizer can create such bundles, Denver has difficulty sustaining more than five micro-ops per cycle or more than three ARM instructions per cycle. By comparison, the maximum sustainable execution rate in ARM mode is two ARM instructions per cycle.

The Denver microarchitecture is a strictly in-order design, eliminating the need for reorder buffers or similar hardware structures. All instruction bundles, whether they come from the ARM decoder or the microcode decoder, are handled in the same way once they reach the scheduler. The scheduler merely checks the micro-ops to ensure that all operands are available; if any operand is delayed (owing to an unexpected cache miss, perhaps), execution stalls until the dependency is resolved. These stalls happen more often in ARM mode, because the optimizer attempts to arrange the microcode to avoid stalls.

The CPU provides 64 integer registers and 64 FP/Neon registers—twice the size of the architected register set. In ARM mode, the ARM registers use only the lower-numbered microcode registers. In native mode, however, the optimizer takes advantage of the wider register fields to address the entire microcode register set, using the extra registers for renaming. This approach is similar to the hardware-based register renaming in most high-end CPU designs, but it eliminates the complex logic needed to assign and track the register mapping.

Microcoded Execution Engine

In number and type, Denver's execution units are similar to Cortex-A15's (see *MPR 5/27/13,* "How Cortex-A15 Measures Up"). Denver has two integer units, only one of which contains a multiplier. Each of the two load/store units can also execute simple integer instructions, providing up to four integer-execution slots per cycle. For full ARMv8 compatibility, each of these units supports 64-bit operations. The FP/Neon units are a full 128 bits wide and can perform two double-precision FP operations or one 128-bit SIMD operation per cycle. Given the CPU's capacity for up to four integer operations, two loads, two stores, and two FP/Neon operations on each cycle, the optimizer has great flexibility in bundling micro-operations.

As with instruction execution, the CPU performs loads and stores in order. It includes a 64KB data cache. The data TLB contains 256 entries in eight ways; each entry maps a standard 4KB page, or two entries can be combined to map a 64KB page. A 2MB level-two (L2) cache backs the primary caches, as Figure 3 shows, and responds in 18 cycles (load-to-use). All the caches have 64-byte lines. An L2 TLB, which Nvidia calls an accelerator cache, has 2K entries in eight ways; it contains address translations as well as partial results from table walks.

The core includes a hardware prefetch unit that Boggs describes as "aggressive" in preloading the data cache but less aggressive in preloading the instruction cache. It also implements a "run-ahead" feature that continues to execute microcode speculatively after a data-cache miss; this execution can trigger additional cache misses that resolve in

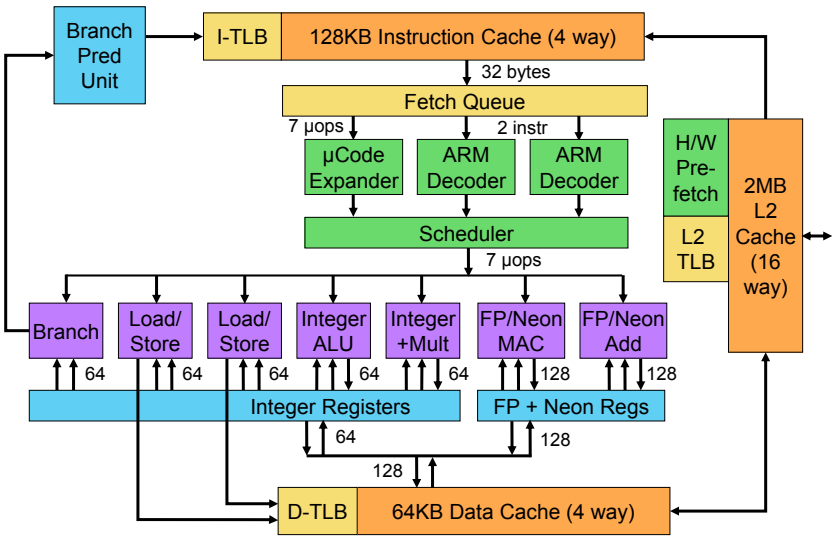

Figure 3. Denver CPU microarchitecture. This design combines a fairly simple front end with a wide set of execution units that includes two integer units, two floating-point units, two load/store units, and one branch unit.

the shadow of the first miss. Once the data from the original miss returns, the results of this speculative execution are discarded and execution restarts with the bundle containing the original miss, but run-ahead can preload subsequent data into the cache, thus avoiding a string of time-wasting cache misses. These and other features help Denver outscore Cortex-A15 by more than 2.6x on a memory-read test even when both use the same SoC framework (Tegra K1).

Although the CPU spends most of its time executing translated code, it must still appear to software as a traditional ARM design. As with hardware OOO designs, interrupts and exceptions pose a challenge. At the start of each translated routine, Denver checkpoints the architected state using internal shadow registers. The caches employ a transactional memory model, which means that store data isn't committed until the end of a "transaction"—in this case, the translated routine (see *MPR 5/26/14,* "Transactional Memory Accelerates"). Nvidia declined to specify the implementation method, but stores can be held in a buffer (a method Transmeta used) or provisionally loaded into the data cache while retaining the ability to invalidate those cache entries if necessary.

At the end of each translated routine, the CPU commits all pending stores. If an interrupt occurs while the CPU is executing translated code, however, the pending stores are aborted and the architected state (including the program counter) is rolled back to the previous checkpoint before transferring to the designated interrupt handler. If the interrupt handler chooses to restart execution, it does so at the beginning of the translated routine.

If an exception occurs during translated code, it is not reported to the exception handler. Instead, the CPU rolls back to the previous checkpoint and aborts pending stores. It then re-executes the offending routine using the original ARM code. If the exception occurs again, the exception handler is invoked; because the ARM instructions execute in order in this mode, no cleanup is required.

Skewing the Pipeline

The Denver CPU uses a moderately deep pipeline with 15 stages, compared with 18 stages for Cortex-A15. Owing to the similarity in pipeline length, it should be able to approach the A15's maximum clock speed. In mobile systems, however, Cortex-A15's speed is limited by its power (thermal limits), not its pipeline. Having only two cores instead of four in the same thermal envelope, Denver offers more thermal headroom, which helps Tegra K1-64 achieve slightly higher clock speeds than the K1-32.

Nvidia chose to "skew" the pipeline, delaying the execute state until after the data cache is accessed, as Figure 4 shows. This arrangement essentially eliminates the load-use penalty, allowing the optimizer to bundle a load operation, a dependent ALU operation, and a dependent store. In this situation, the load/store unit calculates the load address in stage EA2 and accesses the data cache in stages ED3 and EL4. The load data is bypassed directly to the ALU, which executes in stage EE5. The calculation result goes to the second load/store unit, which computes the store address in stage ES6 and writes the store data in EW7.

This design gives the optimizer greater flexibility, allowing it to combine many dependent operations in the same bundle and thus increasing the number of instructions executed on each cycle. Delaying the execution stage, however, extends the time before a branch misprediction is discovered. Instead of potentially executing in stage EA2, a compare instruction executes in EE5. As Figure 4 shows, the misprediction penalty is 13 cycles, which is hefty but still shorter than Cortex-A15's 15-cycle penalty.

To avoid this misprediction penalty, Nvidia put considerable work into improving branch-prediction accuracy. In his presentation, Boggs showed that compared with the A15, Denver reduces mispredictions by about 20% on SPECint2000, SPECfp2000, and Google Octane. For all three benchmarks, the misprediction rate is well below 1%. The company withheld details of Denver's branch-prediction hardware but did indicate that it uses both direct and indirect predictors as well as the usual return address stack.

Tegra K1 Includes Kepler GPU

As it has for other recent processors, Nvidia provided artful renditions instead of actual die photos. These renditions show each Denver CPU occupying the same die area as two A15 cores, and Boggs confirmed that this proportion is roughly accurate. By eliminating one instruction decoder and much of the OOO complexity, Denver should in theory be smaller than Cortex-A15, but several factors tip the scales in the other direction.

The 128KB instruction cache, which accommodates the wide micro-ops, is four times larger than the A15's and consumes considerable die area. The dual 128-bit Neon units are twice as large as the A15's. Don't forget that Cortex-A15 is a 32-bit ARMv7 design, meaning Denver doubles all the integer data paths; this change alone adds at least 20% to the die area. Denver's code translation also requires some incremental logic, including additional tables in the branch unit and hardware checking for run-time problems

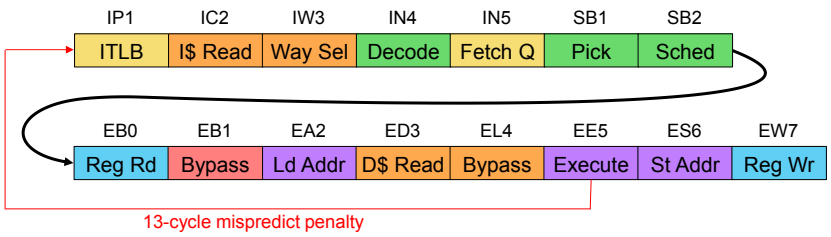

Figure 4. Denver CPU pipeline. The ARMv8 CPU uses a 15-stage pipeline that is similar in length to Cortex-A15's.

that the optimizer doesn't anticipate. Boggs also hinted at additional features to be disclosed later.

Nvidia withheld specific power data, but the two-for-one exchange implies that each Denver core uses about twice the power of Cortex-A15. Power is roughly proportional to transistor count, so this ratio is consistent with Denver's larger die area. Tegra K1-32 (and previous Tegra processors) includes a "companion core" that is optimized for low power on light workloads; the K1-64 omits this core and will simply operate at lower speed and voltage when the workload is light. Qualcomm uses the same approach with its Krait CPUs.

	Snapdragon 805	Apple A7	Tegra K1-32	Tegra K1-64	Snapdragon 810
CPU ISA	ARMv7	ARMv8	ARMv7	ARMv8	ARMv8
CPU Type	4x Krait 400	2x Cyclone	4x A15	2x Denver	4x A57 + 4x A53
CPU Speed	2.5GHz	1.4GHz	2.3GHz	2.5GHz	2.2GHz‡
CPU Perf*‡	3.5 ST / 11.9	5.0 ST / 9.6	3.9 ST / 13.3	6.3 ST / 11.9	4.4 ST / 17.9
GPU Type	Adreno 420	SGX5 MP4	Kepler-192	Kepler-192	Adreno 430
GPU Perf†§	19fps	13fps	27fps	27fps	25fps‡
Video Decode	4K	1080p	4K	4K	4K
ISP	1.0GP/s	Not disclosed	1.2GP/s	1.2GP/s	1.2GP/s
IC Process	28nm HPM	28nm HPM	28nm HPM	28nm HPM	20nm HKMG
First Devices	2Q14	3Q13	2Q14	4Q14 (est)	1H15 (est)

Table 1. Comparison of Tegra K1 products and competitors. The two Tegra K1 versions are very similar except for their CPUs. Both offer better performance than the fastest Apple and Qualcomm processors. ST=single-thread. *Relative to 1.0GHz Cortex-A9; †Manhattan Offscreen 1080p. (Source: vendors, except ‡The Linley Group estimate and §www.gfxbench.com)

In Denver, Nvidia added a new CC4 power state that pulls the CPU's voltage below the minimum operating level but above the minimum voltage for SRAM state retention. The CPU can enter and exit CC4 in only 150 microseconds (depending on the slew rate of the voltage rail), allowing it to use this power-saving mode more frequently. Traditional power gating, in contrast, requires flushing and reloading the caches and registers, so it should be used only when the CPU will be idle for tens of milliseconds at a time.

In most other ways, Tegra K1-64 is identical to the K1-32 (see *MPR 1/13/14,* "Tegra K1 Adds PC Graphics to Mobile"). Both chips include a Kepler GPU with 192 shaders and OpenGL 4.4 support. In the Xiaomi MiPad, the K1-32 delivers 27fps on Kishonti's Manhattan Offscreen test, outscoring Qualcomm's new Snapdragon 805 and Apple's A7 (iPhone 5s) processor, as Table 1 shows. At 4K (UltraHD) resolution, Tegra K1 matches the competition in encoding and decoding video. Using its Chimera ISP, the chip provides leading-edge video and image processing performance.

Industry-Leading CPU Performance

In his presentation, Boggs showed a number of benchmarks that Nvidia ran on both Tegra K1 versions. Denver's best result came on the popular Android benchmark Antutu 4, where the new CPU doubled Cortex-A15's performance in the K1-32. This result allows the company to say the dual-Denver configuration delivers the same performance as the quad-A15 design.

On other benchmarks, however, the K1-64 falls a bit short of the K1-32. On Geekbench 3, a popular cross-platform benchmark, the new CPU delivers 1.6x better performance than Cortex-A15—an impressive result but well short of the 2x needed to make up for the missing cores. Denver does slightly worse on SPECint2000 at 1.4x, but with its double-wide FPUs, it outscores its predecessor by 1.75x on SPECfp2000, as Figure 5 shows. Reviewers of new phones and tablets don't use the SPEC benchmarks, however, although some OEMs favor SPECint2000 as a robust metric of general performance. Taking an average of the five tests shown, Denver outperforms Cortex-A15 by 1.62x, or 1.5x on a clock-for-clock basis.

Apple's dual-core A7 scores lower on multithreaded benchmarks, but its Cyclone CPU beats Cortex-A15 on many single-threaded tests. Tegra K1-64's big advantage is on these single-threaded tests, where the powerful Denver CPU will outperform Apple's Cyclone and stomp all other ARM cores, as Table 1 shows.

Cheekily, Nvidia even compared Denver's performance against that of Haswell, Intel's newest PC CPU. The chart shows the 2.5GHz Denver running neck-and-neck with a 1.4GHz Celeron processor that disables Haswell's multithreaded capability. Thus, Haswell still appears to have a 1.8x advantage in per-clock performance for single-

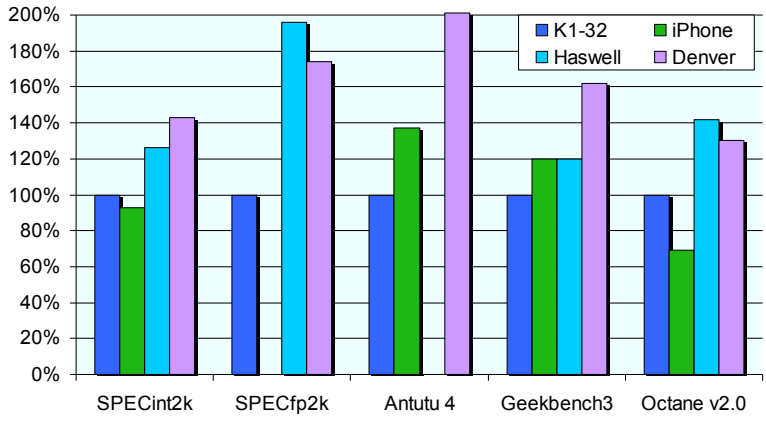

Figure 5. Denver CPU single-core performance. This chart compares a 2.5GHz Denver against a 2.3GHz Cortex-A15 (Tegra K1-32), a 1.4GHz Haswell, and a 1.3GHz Apple A7 processor. (Data source: Nvidia)

threaded code and a greater advantage with multithreading enabled. But to be fair, the dual-core Celeron chip is rated at 15W TDP and carries a list price of $107; Tegra K1-64 should come in at much less than half of those values.

The Snapdragon 810 and other processors with four Cortex-A57 CPUs could exceed Tegra K1-64's total performance, as Table 1 shows. The 810 will use 20nm technology to fit these four 64-bit cores into the same power envelope as the dual-core Denver. Even so, the single-thread performance of the A57 will fall well short of Denver's. We expect the Snapdragon 810, Qualcomm's first high-end 64-bit processor, to appear in smartphones and tablets in 2Q15.

Implications of Dynamic Translation

Dynamic instruction translation has a long history, but it has never seen wide deployment. The technique traces back to research at Bell Labs and IBM in the early 1980s. Some VLIW companies tried it to solve the fundamental problem that multiple VLIW-processor generations cannot execute the same compiled code. Transmeta's products translated x86 instructions into VLIW code, in part to avoid Intel patents on decoding x86 instructions in hardware. The technology is similar to just-in-time (JIT) compilation, which is used for Java and other languages, but dynamic translation compiles from assembly code rather than a high-level language.

Dynamic translation creates some interesting opportunities beyond what standard CPUs can achieve. Although Nvidia declined to reveal any unusual capabilities of the execution units, the native microarchitecture could have features that are absent from the standard ARMv8 architecture, such as fused operations or new addressing modes. The optimizer could easily create these combinations after analyzing the original code and grouping certain operations. This approach could put the "plus" in Nvidia's claimed 7+ instructions per cycle.

Another possibility is that Nvidia could create a version of the Java JVM to translate from Dalvik format directly to microcode. This approach would reduce the overhead of translating to ARM and then retranslating to microcode. Furthermore, the Dalvik code has more semantic information than binary ARM code, enabling greater optimization. Because of Denver's security mechanisms, this JVM would have to reside in microcode format in the optimization cache, reducing space for optimized routines. Nvidia has not developed a direct-translation JVM, but it could in the future.

The company could also provide tools for software developers to incorporate Denver microcode in their apps or middleware (e.g., web browser). This approach would improve performance by reducing translation overhead, although the 3x code expansion would require judicious usage. Perhaps more significantly, it could allow the creation and use of particularly complex instructions that can't be synthesized from ARM code, such as an FFT. After announcing Tegra 3, Nvidia created the Tegra Zone to promote apps optimized for its architecture; it could apply the same approach to distributing microcoded apps. To do so, however, the company would have to expose its microcode format externally and deal with the problem of the format changing in future Tegra products.

Translation can be applied to multiple instruction sets. Nvidia has been working on its first CPU for a long time (Boggs joined the company in 2006, after serving as one of the Pentium 4 architects). At one time, the project was rumored to be an x86 CPU, and later a dual-mode design. Denver could easily execute x86 code just by running a different version of the optimizer; because it lacks x86 decoders, however, it would have to translate every routine before executing it, much as the Transmeta design did; this situation would reduce performance. If Nvidia added x86 compatibility to Denver, it could build chips for Windows 8 tablets or laptop PCs, or even attempt to displace Intel from supercomputers.

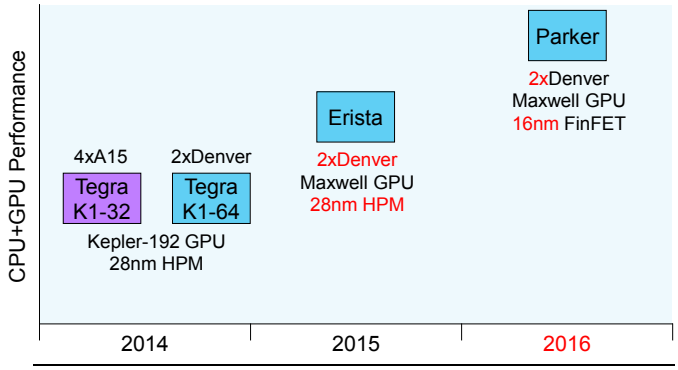

Figure 6. Nvidia Tegra roadmap. The company's public roadmap stops at Erista, but we have added a 2016 release date for the previously disclosed Parker. (Source: Nvidia, except red text per The Linley Group estimates)

Extending the Tegra Roadmap

For now, Nvidia seems focused on Android tablets and other traditional Tegra markets where visual computing is valued. To follow Tegra K1-64, it is already developing a new processor code-named Erista, as Figure 6 shows. The company has released little information on Erista except that it will ship in 2015 and that it will upgrade the GPU to the Maxwell architecture that Nvidia is already shipping into PCs. Regarding performance, the company said only that Erista will add 70% to Tegra K1's Gflops/W rating; this metric likely includes Gflops generated from the GPU as well as the CPU.

To minimize Nvidia's investment, we expect Erista will use the same 28nm Denver CPUs but add the new Maxwell GPU, combined with some minor improvements in the SoC framework. This combination would provide a big bump in graphics performance, helping Nvidia stay ahead of Qualcomm's next-generation Adreno 430 (see

MCR 4/14/14, "Qualcomm Tips Cortex-A57 Plans"). It would, however, do little to boost CPU performance, except for potential gains from improving the optimizer firmware. Still, this simple upgrade should be deliverable within a year. It would follow the same pattern as Tegra K1-32 (Logan), which carried forward the same CPU cores as Tegra 4 while upgrading the GPU.

Nvidia previously disclosed a processor code-named Parker with Denver CPUs and a Maxwell GPU in a 16nm FinFET process. Parker has disappeared from Nvidia's public roadmap, but we believe the company is still working on this project. Delays in TSMC's 16nm process have pushed its production into 2016, necessitating Erista's addition to the plan. Nvidia may also need more time to port Denver to the new process node. The 16nm CPU design could include some minor tweaks to the microarchitecture based on field testing of the initial design. The process shrink should also boost CPU clock speed at the same power level. Alternatively, Parker could implement four Denver cores operating at a lower clock speed; a quad-core design would boost benchmark scores but increase the die cost.

Using the same approach as with Erista and Logan, Nvidia could release in 2017 a second 16nm processor using the same CPU cores as Parker but upgrading the GPU to the recently disclosed Pascal design. Pascal is scheduled to ship in PCs in 2016, so it could be ready for a mobile processor the following year.

Towering Over Its Rivals

Despite Boggs's assertion that dynamic instruction translation is "the architecture of the future," the jury is still out on this question. Dynamic translation didn't cause Transmeta's failure, but it wasn't much help, either. Transmeta's processors scored well on benchmarks but underperformed on Windows code and PC applications, which have many difficult-to-predict branches and fewer frequently repeated routines. To address this problem, the Denver CPU implements several features, including hardware instruction decoders and a hardware prefetch unit, that the Transmeta design lacked. Nvidia says it has tested Denver on a broad range of mobile applications, including web browsing and intense gaming, and found fairly consistent performance.

Tegra K1-64 won't be the first 64-bit ARM processor to reach production; Apple's A7 has been shipping for months, and AppliedMicro's X-Gene recently entered production. The first wave of Cortex-A53 products will probably beat the K1-64 to market as well. But Nvidia will have the first merchant ARMv8 processor for high-end mobile devices. That may sound like a lot of weasel words, but it means Tegra K1-64 will be the only 64-bit option available to makers of tablets and high-end smartphones. This exclusivity will last until Snapdragon 810 launches next spring.

While other processor vendors considered whether to use ARM's Big or Little cores, Nvidia took a different direction, developing a Kong-size core that towers over even Big cores in single-thread performance. Denver's performance per watt is no better than that of its rivals, allowing them to achieve similar chip-level throughput by piling more cores. But most real apps (as opposed to benchmarks) heavily load only one or two cores, so the K1-64 should deliver a far better and more consistent user experience than any other mobile processor. Apple has taken a similar approach with its dual-core iPhone 5s processor, but it uses a much lower clock speed to extend battery life and thus lags the K1-64 in performance.

Nvidia targets a higher power level than Apple and Qualcomm. Only two smartphone models use Tegra 4, and none use Tegra K1; the latter chip appears mainly in tablets, which have larger batteries and tolerate more heat than smartphones. If a phone maker wanted to use Tegra K1, it would probably have to underclock the processor significantly, diluting the performance advantage. But in tablets, Chromebooks, and other larger systems, Tegra K1 has a big advantage over rivals that are designing their CPUs for smartphones. The K1-64 extends that lead by delivering a big jump in single-thread performance and full 64-bit support about a quarter earlier than its primary competitors. These advantages should help Nvidia reverse its declining tablet share, possibly starting with a rumored win in the next Google Nexus tablet. ♦

Price and Availability

Tegra K1-64 is currently sampling and is scheduled to enter production before the end of this year. The company did not disclose pricing; we expect the K1-64 to sell for about $25 in high volume. For more information, see *http://blogs.nvidia.com/blog/2014/08/11/tegra-k1-denver-64-bit-for-android*.